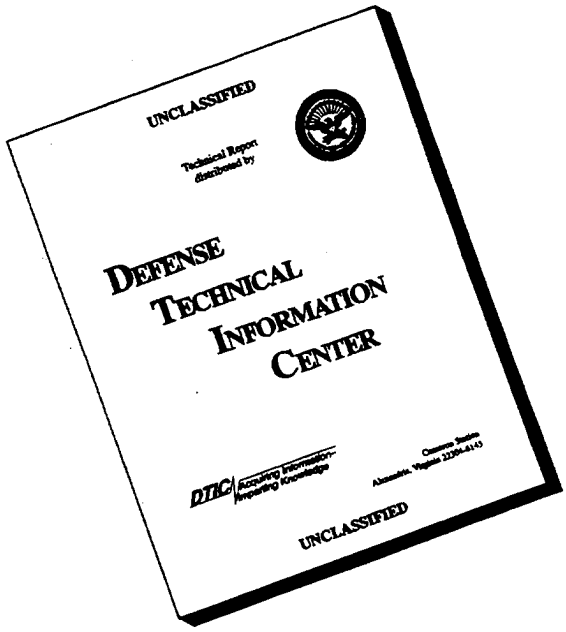
Performance Measurement and Modeling with the Lost Cycles Toolkit

M.E. Crovella, T.J. LeBlanc, and W. Meira, Jr.

Technical Report 580
June 1995

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Performance Measurement and Modeling with the Lost Cycles Toolkit

Mark E. Crovella * Thomas J. LeBlanc
Wagner Meira, Jr.
{crovella,leblanc,meira}@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 580

May 1995

Abstract

Although there are many situations in which a model of application performance is valuable, performance modeling of parallel programs is not commonplace, largely because of the difficulty of developing accurate models of real applications executing on real multiprocessors. This paper describes a toolkit for performance tuning and prediction based on *lost cycles analysis*. Lost cycles analysis decomposes parallel overheads into meaningful categories that are amenable to modeling, and uses *a priori* knowledge of the sources and characteristics of overhead in parallel systems to guide and constrain the modeling process. The Lost Cycles Toolkit automates the process of constructing a performance model for a parallel application by integrating empirical model-building techniques from statistics with measurement and modeling techniques for parallel programs. We present several examples to show how the toolkit facilitates the construction of performance models, and to illustrate the use of the toolkit in solving practical performance problems.

This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-9401142, and ONR Contract No. N00014-92-J-1801 (in conjunction with the ARPA HPCC program, ARPA Order No. 8930). Wagner Meira, Jr. is supported by CNPq, Brazil, Grant No. 200.862-93/6.

*Mark Crovella's current address is Computer Science Department, Boston University, 111 Cummington St., Boston, MA 02215, crovella@cs.bu.edu

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1995	technical report

4. TITLE AND SUBTITLE

Performance Measurement and Modeling with the Lost Cycles Toolkit

5. FUNDING NUMBERS

N00014-92-J-1801 / ARPA Order 8930

6. AUTHOR(S)

M.E. Crovella, T.J. LeBlanc, [illegible] Meira, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester NY 14627-0226

8. PERFORMING ORGANIZATION

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES)

Office of Naval Research
Information Systems
Arlington VA 22217

ARPA
3701 N. Fairfax Drive
Arlington VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

TR 580

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

(see title page)

14. SUBJECT TERMS

parallel performance modeling; analysis; prediction

15. NUMBER OF PAGES

16 pages

16. PRICE CODE

free to sponsors; else $2.00

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

1 Introduction

Parallel programmers are faced with significant challenges in developing efficient programs. Parallel programs often behave in unexpected ways due to the complex relationship between the structure of a parallel program, the machine on which it is run, the number of processors used, the program's input, and the measured running time of the program. As a result, performance tuning of parallel programs is an error-prone and time-consuming process.

Performance prediction offers significant benefits when programming for efficiency on parallel machines. The ability to predict performance helps avoid costly and error-prone experiments that consume valuable resources. Using a performance model of an application, programmers can determine the strengths of competitive implementations, and the scalability of current implementations.

Despite these benefits, performance modeling of parallel programs is not commonplace, largely because of the difficulty of developing accurate models of real applications executing on real multiprocessors. Since parallel overhead can arise from many sources, performance modeling requires an analytical understanding of a wide range of factors spanning hardware and software. Properly and accurately measuring parallel overhead is painstaking and time consuming. The possibility of interactions among factors requires an understanding of how to design experiments to isolate those interactions.

To address these difficulties, a variety of practical performance prediction techniques have been proposed. These techniques can be characterized by their relative reliance on static versus dynamic data. Static methods rely primarily on source code analysis to develop predictions. These methods have the advantage of requiring minimal machine or application measurements, but may sacrifice accuracy due to the limitations of current state of the art in code analysis. As a result, static methods tend to be limited to particular languages, programming styles, or machine types. Within these constraints, several static methods have achieved good accuracy (e.g., [Clement and Quinn, 1993]).

Additional dynamic information is used in training sets (e.g., [Pease et al., 1991]) and template-based approaches (e.g., [Zimran et al., 1990]). Training sets use extensive machine measurements to characterize frequently-occurring instruction patterns. The measurements provide accuracy, but the training sets need to be quite extensive and can be difficult to maintain. Template-based approaches subsume most of the static analysis task in the selection of a program template. Complex performance models are constructed in advance and implicitly selected when a template is chosen. The performance models are cast in terms of basic machine parameters to minimize the need for dynamic measurements. The template-based approach requires the user to match the application to a pre-existing template, and so limits the technique's applicability to specific classes of applications.

Scalability analysis is a completely different approach to performance prediction, where the goal is to characterize combinations of machine and application with respect to their efficiency [Grama et al., 1993]. Scalability analysis develops analytic, asymptotic models of computation and selected overhead categories as a function of the size of the problem and the number of processors. Although these analyses provide insight into the inherent scalability of a particular application and machine combination, they cannot be used directly for performance prediction because of their reliance on asymptotic analysis.

In this paper we describe a toolkit for performance prediction that offers the predictive power of scalability analysis and the accuracy of training sets or templates. In contrast to training sets and templates, our approach is based mainly on dynamic measurements of application programs, so that

our techniques are broadly applicable, and machine- and language-independent. Like scalability analysis, we divide parallel overheads into categories that are amenable to modeling. We call the process of building performance models using these overhead categories *lost cycles analysis* [Crovella and LeBlanc, 1994]. The Lost Cycles Toolkit, which automates the process of constructing a performance model for a parallel application, uses *a priori* knowledge of the sources and characteristics of the overhead categories in parallel systems to guide and control the modeling process. The toolkit integrates empirical model-building techniques from statistics [Atkinson and Donev, 1992; Box and Draper, 1987; Box et al., 1978] with measurement and modeling techniques for parallel programs.

Our primary goal for the toolkit is to offer sufficient ease of use and generality that performance modeling can become a standard tool for programmers during parallel program development. In this respect our work differs from [Brewer, 1995], which also fits empirically measured data to mathematical models, in that his focus is on the use of models to select from among alternative library implementations. Another similar approach is used in [Toledo, 1995], but that work is restricted to SIMD-style applications. To achieve our goal, the prediction method should require as few experiments as possible, the models produced by the toolkit must be accurate enough to be used in a wide variety of scenarios, and the models must apply to all performance factors simultaneously (rather than a single factor, such as the number of processors).

The Lost Cycles Toolkit achieves these goals as follows. First, we define a set of overhead categories for parallel programs that assists in analysis, and provide a tool (`pp`) that measures the overhead in each category at runtime. We ensure efficient experimentation by incorporating experimental design methodology into a tool for automatic experiment generation (`expgen`). We use the performance measurement results obtained from the experiments to select from standard models for overhead categories, using a tool for fitting models of overhead categories to experimental data (`lca`). To ensure accuracy in the resulting performance models, we submit poorly-fitting model components to the user for iterative improvement using the tool `modgen`. The entire process is coordinated via a graphical interface provided by the tool `xmodgen`.

The remainder of the paper describes the Lost Cycles Toolkit in detail, showing first how the toolkit assists in constructing performance models, and then illustrating how those models can be used in performance tuning.

2 Lost Cycles Analysis

Lost cycles analysis [Crovella, 1994; Crovella and LeBlanc, 1994] is an attempt to automate the construction of performance models of an application as a function of runtime factors. The most commonly investigated runtime factors are the number of processors used (p) and the size of the input data (n). However, other runtime factors can have a significant effect on parallel performance, and lost cycles analysis is equally suited to investigating them. For example, the sparseness of an input matrix, the proportion of bright pixels in an image, or the connectivity of an input graph all are significant runtime factors; we have shown in previous work [Crowl et al., 1994] that factors such as these can be quite important in finding the proper implementation for a parallel program.

To model the performance of a parallel program we need to construct expressions for the application's pure computation T_c and parallel overhead T_o. Pure computation is the work the application must perform to solve its problem; it corresponds to the cost of the best serial implementation of the application. Parallel overhead consists of all additional processor-time spent by the parallel

Table 1: Typical Functional Forms for Overhead Categories

Category	Variable	Typical Form
LI	n	$k_1 n + k_2$
LI	p	$k_1 p\sqrt{p} + k_2$
IP	n	k_1
IP	p	$k_1 p$
SL	p	$k_1 \log p + k_2$
CL	n	$k_1 n$
CL	p	$k_1 p + k_2$
RC	p	$\max(k_1 p + k_2, k_3)$

implementation. The basis for constructing a performance model is the formula

$$T(\vec{r}) = \frac{T_c(\vec{r}) + T_o(\vec{r})}{p}$$

in which T is the running time of the application as a function of the runtime factors $\vec{r}$. If $\vec{r}$ consists of only one factor (such a p) then T is a *univariate* model. More commonly, $\vec{r}$ consists of p, n, and perhaps other factors; in this case T is a *multivariate* model.

Values for T_c can be obtained directly from uniprocessor executions, or indirectly from the values for T_o and the running time T. Alternatively, an expression for T_c can be derived using standard complexity analysis augmented by experiments to obtain the constants. Finding a mathematical expression that accurately describes T_o is more difficult however, because overheads can arise in many ways in a parallel system. Lost cycles analysis attacks this difficulty by *decomposition* of parallel overhead. Parallel overhead is decomposed into a set of categories that are intuitively meaningful, mutually exclusive, and complete. The set of categories expresses overheads in common units (ie., lost cycles or LC). Thus, we can directly compare overhead in one category to overhead in another category, since they both correspond to time (ie., cycles) during which no useful work is done. We usually express LC as its sum over all processors in an execution; this sum is equivalent to the value T_o. Although overheads are not independent (so we cannot in general vary one without changing another) the effect of changing LC in an execution is the same no matter what specific kind of overhead is actually changing. Thus, LC forms a single uniform metric for comparing different kinds of overhead.

The decomposition of parallel overhead into categories facilitates performance modeling for two reasons. First, the decomposed overheads can be modeled more easily than the total overhead. Second, each category of overhead has certain characteristic behaviors it often follows, resulting in simple models in many cases. Lost cycles analysis exploits these two observations about decomposed overheads.

The ease of lost cycles analysis arises from the ability to account for most categories of parallel overheads using default models. These default models can be incorporated into a tool for fitting models to experimental data. Examples of the kinds of default models incorporated into our toolkit are shown in Table 1. The table is representative and does not include all typical behaviors; the complete list is embodied in the toolkit, currently with at least three typical behaviors per category.

The overhead categories of interest may vary among machines or applications. The entries in Table 1 focus on the following categories: load imbalance (LI), insufficient parallelism (IP),

synchronization loss (SL), communication loss (CL), and resource contention (RC). We also refer to the remaining time (which represents computation) as RT, and the total execution time as TT.

In previous work [Crovella and LeBlanc, 1993] we showed how to measure lost cycles, using *performance predicates* to specify each category, and a *predicate profiler* to measure the time spent in each category. We have implemented predicate profiling for Fortran programs on the Kendall Square KSR1, C programs on the Silicon Graphics Challenge machine, and Fortran and C programs running under PVM on a network of SUN workstations. The remainder of this paper will illustrate how we use those measurements to develop performance models of applications.

3 Building Performance Models with the Toolkit

In this section we describe how to generate a performance model for a parallel application using the Lost Cycles Toolkit. We use one-dimensional FFT (1D FFT) on the KSR1 as an example application to illustrate each of the steps in model development. In our implementation of 1D FFT we use a 2-stage pipeline, where the first stage consists of input generation, matrix transpose, FFT, and matrix scaling, and the second stage consists of matrix transpose, FFT, transpose, and output checking. Each stage is allocated half of the available processors and uses data parallelism to exploit its processors.

Creation of a performance model for an application using the Lost Cycles Toolkit involves the following steps:

1. The user designs experiments to capture the necessary overhead measurements. Based on user input, the tool `expgen` determines the necessary application executions and creates a program script for execution.

2. The user runs the experiments by executing the script; the predicate profiler (`pp`) automatically extracts the overhead measurements from each execution.

3. Using the program measurements as input, the `lca` tool finds the best-fitting default univariate model for each overhead category. If no default model is sufficiently accurate for a category, the user is alerted by the `xmodgen` interface, and given the ability to suggest an alternative model.

4. Using the `modgen` tool, the user combines the univariate models into the final, multivariate performance model of the application. Sources of inaccuracy in the final model can be traced through the `xmodgen` interface back to particular categories or choices of experiments, allowing the user to adjust models or compensate for noisy data.

As described above, the tool is intended to be used in a highly interactive fashion. At each step in the modeling process, the user may provide information that facilitates or constrains the modeling process, reducing the need for experiments, increasing the accuracy of the models, or improving the descriptive value of the models. Such information may derive from an analysis of the application, or previous experience with application modeling. In those cases where the user cannot provide the appropriate information, the toolkit uses reasonable default settings to produce a model automatically.

3.1 Experiment Design and Generation

In developing performance models from measured data, there is a tradeoff between the time and effort required to gather the data and the accuracy of the resulting models. Our approach is based on the optimum experimental design techniques presented in [Atkinson and Donev, 1992; Jain, 1991].

Our starting point is an instrumented program for 1D FFT. The performance of the program can characterized by two *factors*: P (the number of processors) and D (the size of the input array). The *levels* for these factors (that is, the valid values each factor can assume) are defined by the program and the architecture. Since we use a two-stage pipeline in the implementation of 1D FFT, P can be any even number of processors up to the limit of the architecture. The levels for D as used in the program have the form 2^N, where N ranges from 4 to 9 (i.e., 16, 32, 64, 128, 256 and 512).

The first step in experimental design is to choose the *experiment levels* for each factor; that is, the subset of possible levels for each factor that are representative of the experimental design space as a whole. In general, we should choose at least four levels for each factor, because our formulae are at most second-order (requiring 3 points for a unique fit) and because we want some goodness-of-fit feedback (provided by the fourth point). Additional points give a more accurate measure of goodness-of-fit. For this example, we chose P=[4, 8, 16, 24] and D =[64, 128, 256, 512].

The actual experiments to be performed correspond to some combination of the various levels for each factor. In factorial design [Atkinson and Donev, 1992], an experiment is performed for all possible combinations of the levels for P and D, which would require 16 experiments in our example. While this approach provides broad coverage of the experiment space, it can be very time-consuming. A fractional (or reduced) factorial design attempts to provide reasonable coverage of the experiment space, while requiring significantly fewer experiments. In our case we use a cross-tuple approach based on a subset of the levels for each factor, called *instance levels*. For each instance level, we generate experiments containing that instance level and all other possible levels for the other factors. In our example, this cross-tuple approach requires 12 experiments instead of 16. [1]

In order to generate experiments using the **expgen** tool, the user must supply the factors of interest, and the levels for each factor. The user may choose to supply the instance levels, which determine the actual experiments to be performed, or simply the number of experiments desired. The user can also supply a *replication count*, which determines the number of times each experiment is performed. The toolkit assumes a default replication count of 2; more redundancy in experiments increases the accuracy of the results. From these values, the experiment generation tool (**expgen**) generates the actual experiments.

In our example, **expgen** used instance levels P=[8, 24] and D =[128, 512], and a default replication count of 2. The output of **expgen** is a program script containing the 12 experiments to be performed. Each of these scripts is executed twice (possibly overnight, if the experiments are time-consuming), producing predicate profiling information that is stored for use during model generation.

[1] The number of experiments saved using a reduced-factorial design depends on the number of factors, so the savings could be much greater than in this example.

3.2 Model Generation

We use the measurements produced by the experiment design and generation steps as input to the model generation tool `modgen` to create a model for the application's performance. Model generation takes place in two steps. We first generate a univariate model for each category of overhead that captures the effects of varying each factor in isolation. We then generate a multivariate model for each overhead category that captures the effects of varying all factors in the experimental design space. The model for the application is simply the sum of these overhead models.

Univariate model generation is straightforward. The `lca` tool generates a model for each category of overhead by fitting measured data to standard models of overheads. Our toolkit finds the best model for a category using a direct method of general linear least squares minimization based on Singular Value Decomposition (SVD) [Press et al., 1988]. [2] The metric used for fitting is the determination coefficient (R^2), which ranges between 0 and 1, with 1 being the best possible fitting. The determination coefficient is the fraction of the total variance that is explained by the fit [Jain, 1991].

For each cross-tuple, `modgen` generates a univariate model for each factor by calling `lca`. For example, one of the univariate models we generate describes how load imbalance varies as a function of P when D=128. Another univariate model describes how contention varies as a function of D when P=8. `modgen` determines which models to generate, and calls `lca` to fit a model to a subset of the measured data points.

In multivariate model generation, `modgen` combines and refits the univariate models. Each multivariate model expresses interactions (or lack thereof) among the various factors. If two factors do not interact, then a reasonable form for the multivariate model is the sum of the univariate models. If two factors do interact, then the multivariate model includes the product of the univariate models. `modgen` compares the fit for these alternative combinations of univariate models to the measured data to determine the form of the resulting combined model.

Using this approach, we are able to produce a model for each cross-tuple, by combining all the univariate models for that tuple according to the best fit for their interactions. Since the number of cross-tuples is the product of the number of instance levels for each factor (and therefore usually greater than one), this approach will produce several multivariate models that must be combined into one. We incrementally combine these models as they are generated within `modgen`, using the best fit among the two models or their sum. `modgen` uses all the measured data points in determining this fit.

The user can evaluate and improve the models generated by the toolkit using an X interface produced by the `xmodgen` tool. The toolkit supports hierarchical viewpoints of the performance modeling process; some windows describe the model itself, others describe the detailed information used to generate the model. We illustrate the interface and the hierarchical viewpoints using our FFT example, as shown in Figure 1.

The Fitting Overview window (Figure 1a) is the starting point of the evaluation. This window shows the average determination coefficient for each univariate model representing a factor and overhead category pair (along the top row of values), and for the multivariate model (the bottom row

[2] Direct methods such as SVD are limited to fitting curves that are linear in their coefficients. Thus direct methods can find the values for the coefficients (k's) in expressions such as $k_1 x \log(x) + k_2 x^2 + k_3 e^x$, but cannot typically do so for expressions such as $k_1 e^{k_2 x}$. To solve nonlinear expressions it is usually necessary to use iterative methods, but fortunately the expressions used in modeling parallel overheads are normally relatively simple and are characteristically linear in their coefficients.

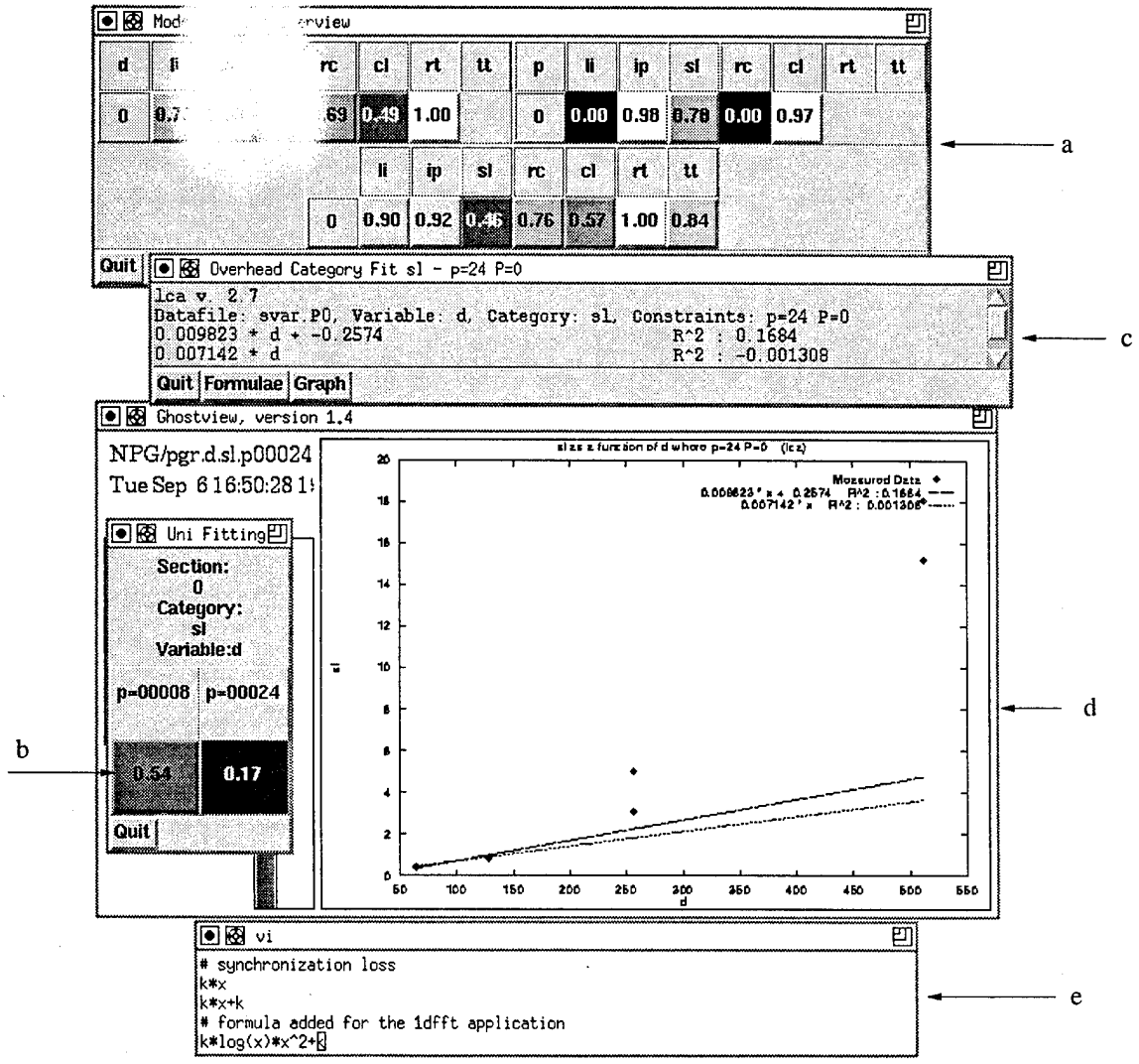


Figure 1: Tool display for 1D FFT illustrating insufficient experiment replication and inadequate default models.

of values). Values close to 1.0 (e.g., insufficient parallelism as a function of D or P, communication loss as a function of P) represent accurate models; smaller values (e.g., synchronization loss as a function of D) represent inaccurate models. (Note that in this display we treat the time remaining after subtracting out all sources of overhead, rt, as simply another category for modeling purposes.)

In this example, the user first wants to know the cause of the low R^2 value for synchronization loss (0.35) as a function of D. Clicking the mouse on the determination coefficient of interest produces the Fitting window (Figure 1b). This window shows the R^2 values associated with each of the instances of P used to generate the univariate model. From this window we can see that the results for P=8 are poor, and the results for P=24 are terrible (0.54 and 0.17, respectively).

Clicking on the determination coefficient button for P=24 in this window produces the Fit window for synchronization loss when P=24 (Figure 1c). This new window shows the two default linear models for synchronization loss, the constants for those models produced by fitting the measured data to the models, and the corresponding R^2 values. From the poor R^2 values, we can conclude that the default models for synchronization loss stored in the toolkit are inadequate for modeling synchronization loss in this application.

Clicking on the "Graph" button at the bottom of this window produces the Ghostview graph (Figure 1d) of these two models and the measured data. This graph not only confirms that the default formulae are inadequate, but also indicates that the measurements exhibit significant variance. The user can reduce the variance by increasing the replication count to 5 (recall the default is 2) and repeating the experiment generation step.

By clicking the "Formulae" button at the bottom of the Fit window, the user can edit the standard models file to include additional models of overhead for a given category. These additional models may be derived by exploiting knowledge of the source code, application structure, or by a visual fit of the measured data points. The editor window (Figure 1e) at the bottom of the figure illustrates this process. In this case, we added the formula $k \times log(x) \times x^2 + k$ based on the observation that the 1D FFT program applies the transform $log(d)$ times to a matrix of size d^2.

After performing a similar type of analysis in each of the cases where the generated model has a low determination coefficient, we can repeat the whole process from the experiment generation step. Continuing with our example, the next iteration of modeling produces the results in Figure 2. As can be seen in this figure, the variance in the data in the graph is now much smaller (as expected, based on the larger replication count), and the new model for synchronization loss accurately captures the measured data. At this point all but one of the determination coefficients are 0.97 or better, and the remaining coefficient (resource contention as a function of P) is 0.93.

The user can use `modgen` to verify the accuracy of the model by having the tool use the model to calculate various measured data points. Once again, we use R^2 as a metric to define how well the model captures the data points used in verification. The user can also use the model to predict values for data points that were not measured.

4 Using Lost Cycles Models

In this section we demonstrate the practical application of our modeling technique to solving problems in parallel programming. Our first example uses performance models to estimate the optimal distribution of workload in a heterogeneous workstation environment. Our second example illustrates the use of our modeling method for predicting the effect of factors other than the number of processors and the data size.

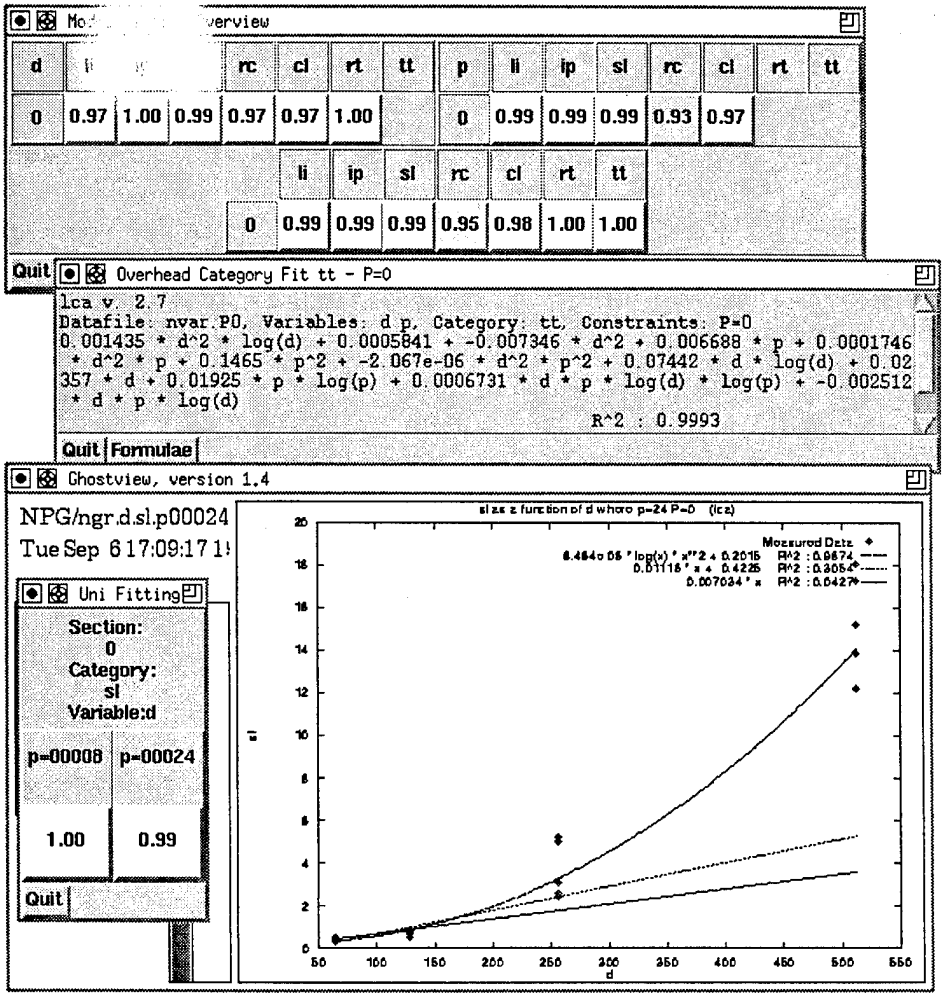


Figure 2: Tool display for 1D FFT illustrating the final stage in the modeling process.

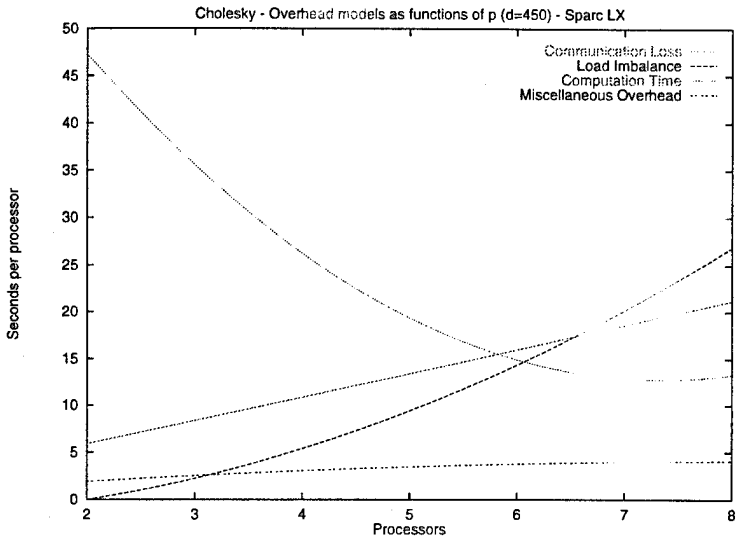


Figure 3: Cholesky: overhead models for Sparc LX

4.1 Load Distribution on a Network of Workstations

Our first example uses models of application running time on a network of homogeneous workstations to predict the optimal load distribution for that application on a network of heterogeneous workstations. Our example application is Cholesky factorization implemented under PVM. Our network consists of Sun Sparcstation 1 and LX workstations. The problem is this: if, at runtime, the program has available an equal number of Sun 1's and LX's, how should the application load be distributed among the workstations? [3]

One obvious solution is to distribute the load in proportion to the estimated speed of the processors in the machines, which is approximately 2/1. Another approach is to use a sequential implementation of Cholesky to benchmark the machines on this application, and distribute the load in proportion to the speed of the machines when executing this application, which is roughly 4/1. (Alternatively, we can use the pure computation category as measured by our predicate profiler to compute this ratio.) Both of these approaches ignore parallel overheads however. Our solution is to develop independent models for the parallel application on Sun 1 and LX workstations, and use these models to calculate the ratio of the speed of the machines on the parallel application as a function of problem size and number of processors.

We first developed models for the running time of Cholesky on a homogeneous set of workstations for both the Sun 1 and LX machines, varying p from 2 to 8, and n (the size of one dimension of the matrix) from 350 to 500. The overhead categories supported by our measurement system under PVM include load imbalance, communication loss, computation time, and miscellaneous overhead related to contention and scheduling effects due to multiprogramming. The models for these individual overhead categories as produced by the toolkit for each type of workstation are presented in figures 3 and 4. From these figures we can see that each overhead category exhibits a similar trend on the two machines, except for miscellaneous overhead, which increases dramatically with an increase in processors on the Sparc 1, but not on the LX.

In this example we are interested in the proper distribution of work between the two classes of machines, rather than in tuning the program to reduce a particular source of overhead on a partic-

[3]It would be straightforward to extend this example to an unequal number of workstations of each type.

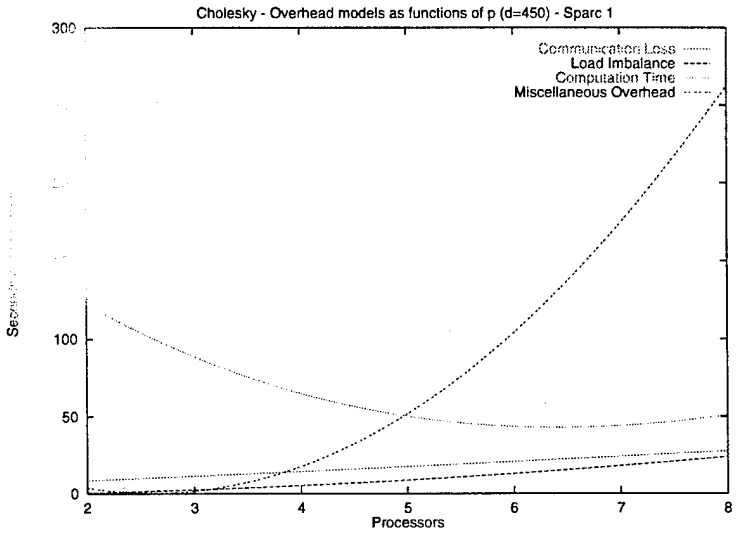


Figure 4: Cholesky: overhead models for Sparc 1

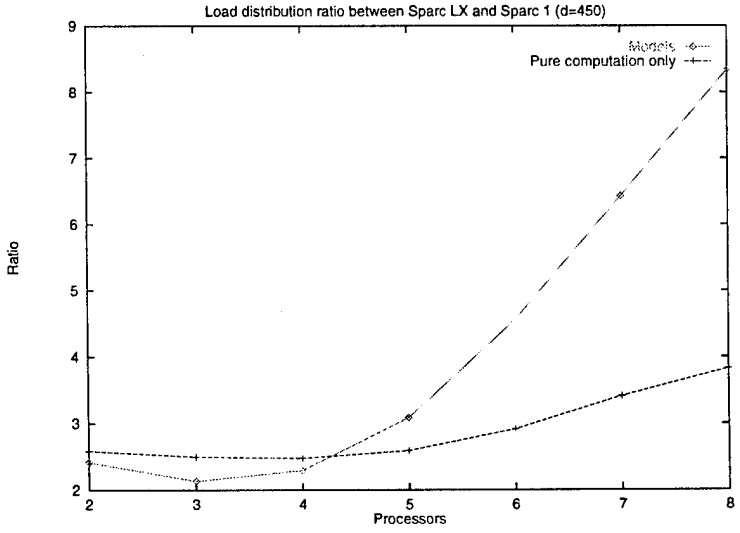


Figure 5: Load distribution ratios

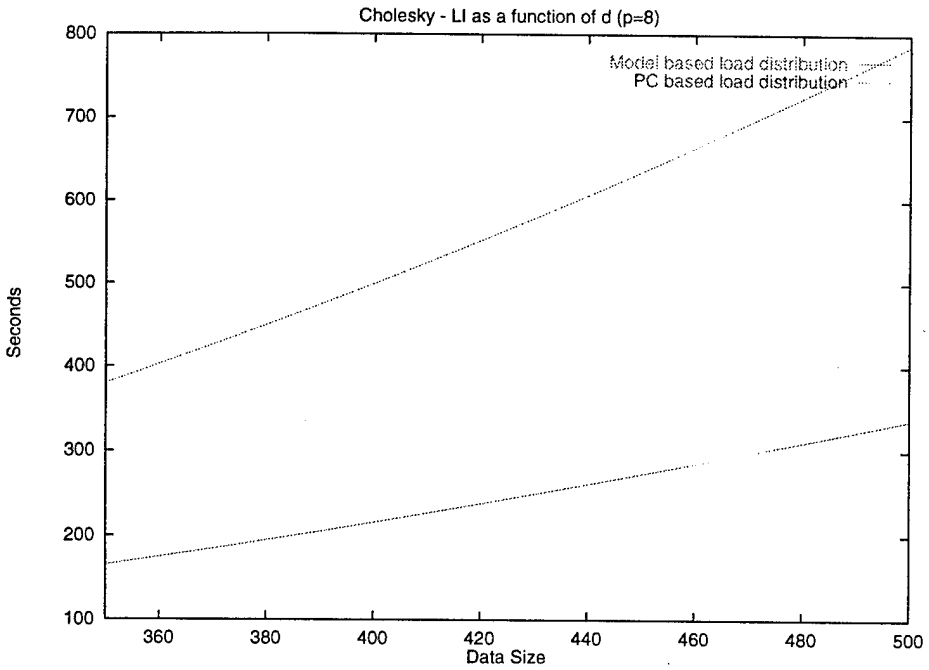


Figure 6: LI using different load distributions

ular machine. Thus, we are interested primarily in minimizing load imbalance on a heterogeneous network. Since we are dealing with only two classes of machines, we can express the distribution of work as a ratio (workload for LX/workload for Sparc 1). We can calculate this ratio using only the pure computation time measured during an execution of the application or using the overhead models produced by the toolkit. In figure 5, we present both ratios as a function of the number of processors, while fixing d=450. In comparing the load distribution suggested by the ratio of computation times for the application to the ratio provided by the models, we see that the models suggest a comparable ratio for a very small number of processors, but a much higher ratio of work for the LXs (8/1) when $p = 8$.

To verify this result, we executed the application using both ratios on a network of 4 Sun 1's and 4 LX's. The workload distribution provided by the models produced an improvement in running time of 25-40% depending on the data size. The reduction in load imbalance as a function of data size is show in figure 6. As seen in the figure, load imbalance decreases by more than 50% for large data sizes, when using the models generated by the toolkit to determine the workload distribution.

It is worth noting that the models for a homogeneous network of workstations are fairly accurate at predicting the running time of the application, and can be used to solve the problem of workload distribution, even though we did not fully explain all the cycles lost to miscellaneous overhead. Whether the dramatic increase in this overhead category on the Sun 1's that results from an increase in processors is attributed to contention or scheduling effects isn't important for the purposes of solving the problem of workload distribution with our modeling approach.

4.2 Computation Time in Speculative Search

In earlier work [Crowl et al., 1994] we studied the performance of several alternative implementations of subgraph isomorphism on a variety of shared-memory multiprocessors. In that work we

performed over 37,000 experiments to show how the number of isomorphisms sought, the number of available processors, and the underlying architecture all affect the choice of an efficient parallelization. Here we use our modeling technique to predict the amount of time processors spend searching for a solution on a KSR1.

Our parallel pro[illegible] is based on Ullman's sequential tree-search algorithm for subgraph isomorphism [Ullman. [illegible]hich successively postulates a mapping from each vertex in the small graph to a vertex in [illegible] graph. Each mapping constrains the possible mappings for succeeding vertices of the sma[illegible] The process of postulating mappings for vertices in the small graph continues until an i[illegible]sm is found, or until the constraints preclude such a mapping, at which point the algorithm [illegible]es a different mapping for an earlier vertex.

There are many [illegible] exploit parallelism in the search for an isomorphism. In this example we focus on the coarsest granularity of parallelism, which occurs in the tree search itself. We search each subtree of the root node in parallel, using depth-first, sequential search at the remaining levels. This type of parallelism is *speculative*, in that we might not need to search multiple subtrees of the root in parallel in order to find a solution quickly. However, if the solution space is sparse, we might benefit from searching several subtrees in parallel.

In our experiments input graphs are randomly generated from four parameters: the size of the small and large graphs, and the probability for each graph that a given pair of vertices will be joined by an edge. We use S for the number of vertices in the small graph, L for the number of vertices in the large graph, and s and l, respectively, for the edge probabilities in each of the graphs. The total number of leaves in the problem space tree is

$$\binom{L}{S} S!$$

This number is very large; for $S = 32$ and $L = 128$ (the size of our experiments), there are approximately 4×10^{65} leaves. To avoid searching among all the possible leaf nodes, we apply a set of *filters* at each node in the search tree. These filters prune the search space enough to make the problem tractable.

In our earlier work, we characterized the input graphs (and the difficulty of finding a solution) using a *density* function. We define the density d of the search space to be the probability that a randomly-selected leaf will be a solution:

$$d \equiv (1 - s + sl)^{S^2},$$

where S^2 is the number of potential edges in the small graph and $1 - s + sl$ is the probability that a given potential edge will be successfully matched. Our goal here is to discover how the time processors spend looking for a solution varies with density when searching for a single isomorphism.

We attempted to use lost cycles analysis to develop a model of pure computation for the tree-parallel implementation of subgraph isomorphism as a function of density. Our initial attempt was not successful however, because we could not find an analytic form for pure computation that was linear in its coefficients and produced a good fit to the measured data. This empirical evidence that computation time is not a simple function of density has an intuitive explanation: density only captures the prevalence of solutions in the leaf nodes of the tree, whereas computation time depends greatly on the effectiveness of the filters used for pruning (for which we have no analytic formula). Further evidence for the unsuitability of density as a predictor of computation time comes from an examination of the experimental data, wherein different pairs of s and l (the edge probabilities for the two graphs) can produce the same density, but still result in very different computation times.

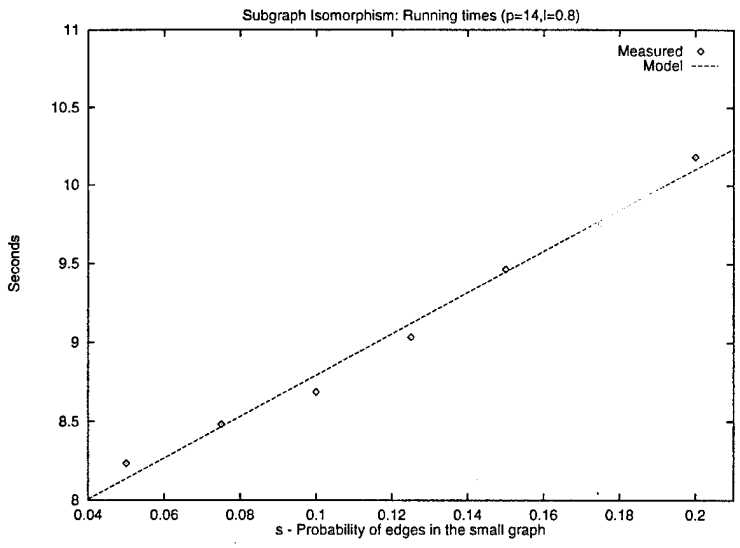


Figure 7: Subgraph Isomorphism: Computation time for various s probabilities

A closer examination of the experimental data shows that computation time *can* be consistently predicted as a function of s and l. Therefore, rather than model computation time as a function of density, we modeled it as a function of s and l (which are used to calculate density). In our experiments we varied s between 0.05 and 0.2, l between 0.8 and 0.95, and p between 8 and 14. [4] The best fit of computation time as a function of s is shown in Figure 7. The fitting for computation time as a function of l is also very good.

This example illustrates several strengths of our approach. First, we were able to model computation time as a function of variables other than p, the number of processors, and n, the size of the input data. Second, we were able to use our modeling technique to discover that density was an inappropriate measure to calculate computation time, but that the values used to calculate density (s and l) were an appropriate determining factor of computation time. Third, we were able to create a multivariate model of computation time (as a function of s and l), which was required to capture the complexity of the input. Finally, we were able to develop an accurate model of computation time (and a good model of running time) without an analytic form for the efficacy of filters.

5 Conclusion

Performance prediction can be a very useful tool for parallel programmers, offering assistance in the difficult task of tuning and evaluating a program's performance and scalability. Unfortunately a number of obstacles have prevented programmers from routinely building performance models of their programs. These obstacles include the difficulty of measuring the many potential sources of inefficiency in parallel systems, the challenge of describing those overheads mathematically, and the requirement to design experiments that yield models of suitable accuracy.

[4] We limited the range of values for s and l to keep the time spent on experiments manageable. This application depends on a random number generator to create the graphs, so each experiment has to be replicated many times with different seeds to ensure average case behavior. Much larger values of s or smaller values of l would significantly increase the difficulty of the search, and the running time of each experiment. In any case, the values we use here cover the range between sparse and dense solution spaces in our earlier paper.

The Lost Cycles Toolkit represents an attempt to make performance prediction available to parallel programmers. It does so by automating as much as possible of the performance modeling process. The toolkit automates experimental design, performance measurement, univariate modeling, and multivariate modeling. It also provides feedback to the user on the quality of the resulting model, and allows the user to trace inaccuracies back to the component models and performance data. This potentially iterative model refinement means that the tools can aggressively automate processes like model selection, since the user is alerted whenever a tool's decision or assumptions are inaccurate.

In this paper we have described the methods and tools comprising the Lost Cycles Toolkit, and we have shown how they are used in practice. In addition, we have shown examples describing the concrete benefits derived from using lost cycles analysis to build performance models of applications. These examples demonstrate the utility of the toolkit, but several caveats are in order.

- Accurate models require many experiments. The toolkit provides an instrumentation package that, at least for Fortran programs, hides the complexity of instrumentation, and also guides the user in the judicious selection of experiments, and automates (via program scripts) the experiment process. Nonetheless, the experiment phase can be time consuming and resource intensive, and may not be practical for long-running applications.

- The set of default models embedded in the toolkit for each overhead category may not accurately characterize the behavior of a particular application, and the user may not have the expertise required to suggest an alternative model based on an analysis of the program. In the worst case, the toolkit can automatically select the best choice from the default models, or the user can suggest an alternative based on a visual inspection of the experiment data points. We are considering adding source code analysis to the toolkit to assist the user in selecting default models for overhead categories.

- The predicate profiler may not be able to capture all the categories of interest in every environment. On the KSR we exploit performance monitoring hardware to isolate communication and contention costs; on the SGI machine we combine these two categories and measure them indirectly by comparing actual execution time to emulated execution time on a perfect memory system (using the *pixie* tool).

Despite these caveats, we have found the Lost Cycles Toolkit to be useful for performance modeling. In addition, a number of improvements are under way or in the planning stage. The extension of lost cycles models across performance boundaries created by system effects (cache and physical memory sizes) represents an open problem. We have used lost cycles analysis on runtime factors other than n and p, but more work is necessary to allow automatic model selection for those factors. Finally, we feel that the extension of the lost cycles methods to portions of programs, rather than complete applications, seems a promising approach to the problem of application restructuring.

References

[Atkinson and Donev, 1992] Atkinson, A. C. and Donev, A. N. (1992). *Optimum Experimental Design.* Oxford Statistical Science Series. Oxford Science Publications.

[Box and Draper, 1987] Box, G. E. P. and Draper, N. R. (1987). *Empirical Model-Building and Response Surfaces.* Wiley Series in Probability and Mathematical Statistics. John Wiley and Sons, Inc.

[Box et al., 1978] Box, G. E. P., Hunter, W. G., and Hunter, J. S. (1978). *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building.* Wiley Series in Probability and Mathematical Statistics. John Wiley and Sons, Inc.

[Brewer, 1995] Brewer, E. A. (1995). High-level optimization via automated statistical modeling. In *Proceedings of PPOPP 95.*

[Clement and Quinn, 1993] Clement, M. J. and Quinn, M. J. (1993). Analytical performance prediction on multicomputers. In *Proceedings of Supercomputing '93*, pages 886–894.

[Crovella, 1994] Crovella, M. E. (1994). Performance prediction and tuning of parallel programs. PhD Dissertation TR 573, Dept. Computer Science, University of Rochester, Rochester, New York.

[Crovella and LeBlanc, 1993] Crovella, M. E. and LeBlanc, T. J. (1993). Performance debugging using parallel performance predicates. In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 140–150.

[Crovella and LeBlanc, 1994] Crovella, M. E. and LeBlanc, T. J. (1994). Parallel performance prediction using lost cycles analysis. In *Proceedings of Supercomputing '94*, pages 600–609.

[Crowl et al., 1994] Crowl, L. A., Crovella, M., LeBlanc, T. J., and Scott, M. L. (1994). The advantages of multiple parallelizations in combinatorial search. *Journal of Parallel and Distributed Computing*, 21(1):110–123.

[Grama et al., 1993] Grama, A. Y., Gupta, A., and Kumar, V. (1993). Isoefficiency function: A scalability metric for parallel algorithms and architectures. *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*, pages 12 – 21.

[Jain, 1991] Jain, R. (1991). *The Art of Computer Systems Performance Analysis.* Wiley and Sons, Inc.

[Pease et al., 1991] Pease, D., Ghafoor, A., Ahmad, I., Andrews, D., Foudil-Bey, K., Karpinski, T., Mikki, M., and Zerrouki, M. (1991). Paws: A performance evaluation tool for parallel computing systems. *IEEE Computer*, pages 18–29.

[Press et al., 1988] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. (1988). *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press.

[Toledo, 1995] Toledo, S. (1995). PerfSim: A tool for automatic performance analysis of data-parallel Fortran programs. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, Virginia.

[Ullman, 1976] Ullman, J. R. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM*, 23:31–42.

[Zimran et al., 1990] Zimran, E., Rao, M., and Segall, Z. (1990). Performance efficient mapping of applications to parallel and distributed architectures. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II–147 – II–154.